UCRL-CONF-203192

LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

Practical SQE on a Large Multi-Disciplinary HPC Development Team

J. Robert Neely

March 26, 2004

International Workshop on Software Engineering for High Performance Computing System (HPCS) Applications
Edinborough, United Kingdom
May 23, 2004 through May 28, 2004

Practical Software Quality Engineering on a Large Multi-Disciplinary HPC Development Team

Rob Neely
Lawrence Livermore National Laboratory, Livermore, CA, U.S.A.
rneely@llnl.gov

Position Paper for the
International Workshop on Software Engineering for High Performance Computing System Applications

1. Introduction

In this paper we will discuss several software engineering practices that have proven useful in a large multi-disciplinary physics code development project at Lawrence Livermore National Laboratory.

In the project discussed here, as with many large scale efforts in HPC scientific computing, we have had to balance the competing demands of being a stable "production" code that our user base can rely on with being a platform for research into new physics, models, and software architectures. Much of this has been learned through necessity and experience. Likewise, much of it has been learned through interactions with other similar projects and hearing of their successes, and tailoring their ideas to our own requirements.

The ideas presented here are not meant to necessarily transfer to other environments with different needs. It is our belief that projects need to be given large latitude in defining their own software engineering process versus a prescribed a solution. However, the ideas presented here are hopefully high level and general enough that we hope other projects might find some inspiration and adopt similar methods if it is to their benefit, much as we have done through the years.

2. Project Background

This project involves a code which has been around in one form or another since the mid 1980's, at which time it was run primarily on the Cray vector architectures. In the early to mid 1990's, an extensive rewrite was undertaken with several primary goals: 1) Porting to MPP architectures and a message passing base, 2) rewriting the high level architecture in a more modern language with better support for data structures and dynamic memory, and 3) building a code which could act as a framework for implementing new physics while not adversely disrupting the core validated physics. The team simultaneously grew in size from a handful of people to a large multi-disciplinary team consisting of code physicists, engineers, chemists, computer scientists, and analysts.

For many years our laboratory, software quality engineering has been something that is left up to the individual projects to define. On this particular project, we took the attitude that anything that made our lives easier by a) automating (and thus simplifying) the workflow, b) reducing the number of phone calls from our users, c) making it easier to add new capabilities to the software, or d) generally allowing us more time to work on "fun stuff", was considered good software engineering. Processes which seemed arbitrary or generally not applicable to our environment were not considered, or ejected. As a result, we have been steadily working toward a software workflow process which we believe is (asymptotically) approaching optimal for our particular environment. Below we will touch on just a few of the areas where we have seen a positive impact.

3. Regression Testing

The value of regression testing (testing answers produced by a given version against a set of validated baselines) first becomes apparent to developers when they discover that someone else has inadvertently broken their software. Although frustrating, it's hard to cast blame when the other person can simply say "But how was I supposed to know - you never added a test problem which would let me know I broke it!". So if for no other reason, regression testing makes for a good defensive strategy!

On our project, we are in the process of defining several levels of regression testing, all under the framework of a set of customized scripts which will automate the process. These are all designed to catch bugs as early in the development cycle as possible. They are:

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

- "Smoke" test
    - Prevent bugs from ever making it onto the integration branch
- "Overnight" test
    - Catch bugs missed by the smoke test within 24 hours
- "Promotion" test
    - Catch bugs before they make it into a public release of the software

Our most often run test suite is the aforementioned "smoke test". The main purpose of this test is to make sure that no one is allowed to commit a change into our source code repository which would break something that worked prior to the commit. The smoke test suite is a large set of problems (> 100) that attempt to maximize coverage in a minimal amount of time (preferably 30-60 minutes total, or 10-60 seconds each) but running small, low resolution, fast running “toy” problems. We keep a set of baseline answers in the form of a restart file from which we do a bitwise comparison the end of each run. If the answers have changed for any reason, this must be documented and the baseline answers in the repository replaced. The smoke test can run some problems in parallel, although we're usually limited to just several processors of a shared memory workstation for simplicity.

We have stressed the importance of bitwise comparison (versus a differencing that allows for relative or absolute tolerances) in the smoke test because we have found that often many bugs manifested themselves in very seemingly minor ways, and could easily be either brushed off as "round-off error" without careful study, or hidden by tolerances. So although we don't require developers to always be able to bitwise reproduce answers (as obviously this would preclude desirable activity such as algorithm improvement or certain performance enhancements), we do want them to understand and document the reasons for any changes, no matter how minor. Furthermore, bitwise comparison of binary restart dumps are made easier through the usage of a portable binary format (similar to HDF5 [1] in functionality) which provides a diff tool from which we can easily filter out differences that do not matter, such as timestamps or code version numbers embedded in the file.

In a perfect world, our smoke tests would catch all bugs being introduced into the code. However, some bugs just will not manifest themselves in small, short running problems. Thus, our second tier of testing (what we call our "overnight" test) adds another layer of test problems designed to as quickly as possible (within 24 hours of a commit) determine if a bug has been introduced to the repository. It is different from the smoke test suite in two primary ways: First, it runs for a longer time, as unlike the smoke test, no one is typically waiting on the results to continue the process of updating the code. As the name implies, we run this suite of jobs during off-hours, and a report is waiting for us each morning if a failure has occurred. Secondly, the tests tend to stress more integration of multiple features of the code. They are often tests contributed by our users, or designed by developers specifically to stress attributes of the code that make for a longer runtime. For these tests, we check for differences in a representative set of values (e.g. min/max coordinate extends, min/max pressure, etc...) for comparison with baseline results.

Our final tier of testing is our "promotion" test suite. The promotion test suite consists of a set of problems that typically will take days to run, and are large scale parallel multi-physics problems. Our project supports a concept of multiple branches of development (see "configuration management"), and this test is a final check to make sure that typical user problems which ran on past stable versions of the code continue to run. Likewise, this indicates that the months of new development is ready for release to users.

In all cases, we try to continuously improve the quality of each suite. If a bug snuck into the repository that should have (or could have) been caught, we will either add a new test problem or modify an existing test problem to one of the suites to make sure it does not happen again.

4. Configuration Management

A well-defined software configuration management (SCM) system has been key to both defining and enforcing our software process workflow. At the core of our SCM system is the concept of maintaining multiple simultaneous branches of the code. Inspired by projects such as the Linux kernel which maintain a stable bug-fix branch while simultaneously doing major development on a different branch - this technique has been an important mechanism for keeping a stable code base for our users while also allowing them access to the latest features in the form of a beta version of the code. We typically aim to release a new "main" (stable) version of our code approximately every 6-9 months, although in reality the decision is made based on a combination of what we find our users using (i.e. have they all jumped from the main to beta version?), and what “feels right” based on project management decisions. In the interim, the developers are working daily on our "maindev" branch to keep it as stable as possible, while applying bug fixes to the main branch as necessary.

Before creating a new main branch, we feature freeze the maindev version and focus entirely on reducing bugs and enhancing stability (this is where we run our complete set of "promotion" tests). This phase is called "premain", and can last anywhere from 2-6 weeks. During this time, we begin the next maindev branch - which effectively leaves us with three simultaneous branches of development: the main stable branch, the premain branch (the candidate for the next main release), and the next maindev branch.

For development where we anticipate potential large disruption to the code base, we try to get those in early on a new maindev branch so as to give ourselves plenty of time to work out the potential bugs before users begin demanding the features on that branch, and thus a public release.

The details of how all this is performed are beyond the scope of this paper, and similar techniques can be found in other literature [2]. However, an important aspect to making this rather complex usage of an SCM system work has been the introduction of a set of customized scripts designed to ease the process for developers - many of who are not computer scientists by training, and simply want an SCM system to stay out of their way. By providing these scripts to encapsulate our process, we help make sure that developers don't "cheat" by doing an end around to avoid repetitive or onerous tasks. But it also helps that they find SCM something simple to use, unobtrusive, and ultimately indispensable.

5. Risk-Based Software Quality Assurance

Our lab is nearly completed drafting its "Institutional Software Quality Assurance Policy", which is designed to cover all software projects on site. As a U.S. Department of Energy (DOE) National Laboratory, we have a terrific range of software development going on in the area of High Performance Computing ranging from small single person "research codes", to larger scale simulation codes being used for U.S. nuclear stockpile stewardship. How can an institutional policy apply to such a wide array of projects? And how can we ensure our researchers that they won't be swept up into a frenzy of CMMi [3] standardizations?

We believe the answer lies in applying a risk-based approach to SQA. In our institutional policy, we have outlined a rather simple questionnaire designed to help each project determine its risk level: low, medium, or high. These levels are determined by investigating a combination of consequences of software failure, and likelihood of failure. The details are not important, but high risk roughly maps to great loss of money, reputation, or human life. Low risk maps to inconsequential results of failure such as inconvenience to the user. Mid risk obviously falls somewhere in between, and applies to a great number of long term funded projects.

Once the level of risk for a project has been determined, our policy goes on to suggest a level of formality for a series of standard SQA topics. For example, a low risk project may only need to perform requirements management on an informal, ad-hoc scale. While a high risk project is expected to conform to more formal standards as appropriate.

The policy does not go so far as to prescribe how any of the suggested methods should be applied. It has long been the understanding at our laboratory that one size does not fit all, and that prescribing anything as a mandate (be it a tool, method, or process) will result in push back from the staff. The only requirements issued by the policy are that each project must do the risk analysis to determine their level of risk, and they must reassess their risk at regular intervals. (ie - a project that starts out as a low-risk research project may eventually find itself being used for safety analysis, and thus mid or high risk).

6. Metrics for Expected CPU Performance

Any HPC project will ultimately be expected to quantify their performance on whatever hardware the funding institution has purchased. Given that percentage of peak performance is not a good metric on which to base the success or failure of a particular code to optimally use architecture, we have done some very preliminary work to help us understand what sort of performance we might expect from a given CPU architecture.

For example, consider the IBM Power3 CPU, which has a peak speed of 1.5 GFlops. In order for a code to obtain this peak performance, it must issue two fused floating point multiply-add (FMA) instructions every cycle, for a total of 4 flops per cycle (4 x 375 MHz = 1.5 GFlop).

What if your application is not able to issue 100% FMA instructions, but is reliant on regular add and multiply instructions? What if it can't feed the CPU with data fast enough to issue two floating point operations per cycle? What if your code is dominated by integer calculations? Obviously the answer is you'll get much lower than peak. But what can we expect?

To help answer this question, we took measurements from several of our codes using the freely available PAPI [4] tool. For this study, we focused on two primary metrics:

1) the floating point instruction mix (the percentage of floating point instructions that were FMA instructions), and 2) the *computational intensity*, or the number of floating point instructions issued per memory access.

Given just these two algorithmic characteristics of our code, we then devised a very simple model which helped us put an upper bound on expected performance from the Power3 architecture. With this model, we were able to show that assuming the metrics collected are a given (i.e. we can't change the instruction mix or computational intensity, at least not dramatically) we could expect anywhere from 16% to 37% as an upper bound on performance for the Power3. This did not yet even take into account other factors such as cache misses, integer instructions, branch mispredictions, etc…

Given this model, it was much easier to go to our reviewers and show that although we were only getting approximately 10% of peak CPU performance on this architecture – we were within striking distance of this theoretical upper bound.

7. Software Architecture and Refactoring

A sound software architecture is clearly key to the longevity of a software project which is constantly undergoing changes in requirements. Although we do not claim to have the most modern or robust architecture in the HPC world, we are slowly improving on our legacy code roots by incrementally introducing new and improved data structures, programming languages, 3rd party libraries, etc…

Although we don't feel like we have any particularly special magic for managing this sort of process, we are staunch believers in disciplined restructuring and rewriting of code for the sake of improving the underlying architecture. A key element of this technique (often referred to as *refactoring*) is that management must accept that a potentially long period of time may be spent in this process with no obviously apparent benefit to the end user. In other words, when refactoring is done – no new features will have been added to the software that would be immediately observable. Of course, longer term – good refactoring will add extensibility to the code and allow for changing requirements to be easily implemented without layering "hack upon hack".

This sort of "rebuilding the plane while in flight" makes for difficult project management, and begs for several of the techniques outlined elsewhere in this paper. In particular, a good regression test suite is essential if one is going to make substantial changes to the software architecture with the expectation that answers should not change. Likewise, some sort of multi-branch development that isolates potentially disruptive changes from a stable version of the code that users are relying on is highly desirable.

In the past, the problem of codes "collapsing under their own weight" was often solved by performing a large rewrite. While this model worked well in the past, many projects are now much more complex (3D, multiphysics, highly parallel, etc…) and as such, we cannot afford to simply rewrite them every 5-10 years. Some codes are now expected to last decades, and as such – a positive attitude toward refactoring is an important project management consideration.

8. Other Useful Metrics

In our never-ending quest to improve our software processes (not to mention prepare ourselves for the inevitable SQA audit!), we are currently investigating other metrics that we believe will hold some promise. We are also finding new ways to apply metrics we have gathered all along in new ways.

Test function coverage data is something that we have been gathering for quite some time, and it has been useful in determining where our various test suites are lacking. Likewise, it has been useful in helping us design our smoke test suite in that we can easily sort our entire suite of problems on a chart with function coverage on one axis and runtime on another. By doing this, we can quickly determine a set of tests which will give us the most "bang for the buck", or the most coverage in the shortest amount of time. (Recall that a critical requirement of our smoke test was a short execution time, as it determines the rate at which developers can update our mainline branches).

Code complexity, as determined through static analysis tools such as McCabe QA[5], is something else we are beginning to use in a new process. Although we have gathered various complexity metrics from McCabe through the years, they were often used primarily as a way to target coding for refactoring. Now we are looking into using these metrics in combination with our test coverage metrics to improve our test suites. The idea is to make sure that the most complex parts of the code are exercised more often.

The final metric we will discuss here is the "user satisfaction" metric. Gathered through an interview process by either the project management, or the teams' SQA authority, this involves a simple questionnaire with a "rate this 1 to 10" section covering several high level areas such as reliability, usability, flexibility, support, etc…

User satisfaction is really what doing good software engineering is all about, is it not? But how often do we quantitatively ask our users how they think we're doing? This metric is almost too obvious, yet we found that we were pioneers in using it, at least in our circle of projects.

Our first round of surveys did not lead us to any surprises in that we scored reletively lower in areas where we suspected our users would have reason to score us low. But the real value in this metric may not become apparent for another several years when we can begin to detect trends over time.

Of course, in order for this sort of metric to be statistically valuable, one must choose a representative set of users. This includes both the happy users, and the perpetually angry users!

9. Conclusion

Developing software in the world of high performance computing is unique and distinct from industry standards. We often have ill-defined and changing requirements, developers with a lack of expertise in software engineering, customers who are either themselves developers or sit right down the hall, as well as other key differences. Each HPC project needs to consider carefully what software quality engineering processes they will put in place, with a goal toward making their lives easier, and making the end customers happy.

Although we don't claim to have all the answers, we do have a relatively large HPC software project that has managed to pick and choose (and sometimes self-discover) software engineering concepts that have proven most valuable to us. The most valuable metric however, is one that probably cannot be easily measured: the sense from the development team that it is working well together, and the users are happy with the results.

Some of our most valuable processes have come from conversations with distant colleagues in conversations that started out like "We have a problem with process X, how do you deal with that?" It is rare that a process torn from the pages of a software engineering textbook has proven valuable to us. We hope this paper serves as guidance for perhaps other groups beginning to struggle with growing pains. Likewise, we hope to learn from other groups what they consider to be their best practices so we can steal them for ourselves!

10. References

[1] http://hdf.ncsa.uiuc.edu/HDF5

[2] "Software Configuration Management Patterns: Effective Teamwork, Practical Integration", Stephen P. Berczuk, Addison-Wesley, 2003.

[3] http://www.sei.cmu.edu/cmmi

[4] http://icl.cs.utk.edu/papi

[5] http://www.mccabe.com/iq_qa.htm